Exploring High Throughput Computing Paradigm for Global Routing

Yiding Han, Dean Michael Ancajas, Koushik Chakraborty, and Sanghamitra Roy

Electrical and Computer Engineering, Utah State University
{bitcars, dbancajas}@gmail.com, {koushik.chakraborty, sanghamitra.roy}@usu.edu

***Abstract*—With aggressive technology scaling, the complexity of the global routing problem is poised to rapidly grow. Solving such a large computational problem demands a high throughput hardware platform such as modern Graphics Processing Units (GPU). In this work, we explore a hybrid GPU-CPU high-throughput computing environment as a scalable alternative to the traditional CPU-based router. We introduce *Net Level Concurrency (NLC)*: a novel parallel model for router algorithms that aims to exploit concurrency at the level of individual nets.**

To efficiently uncover NLC, we design a *Scheduler* to create groups of nets that can be routed in parallel. At its core, our *Scheduler* employs a novel algorithm to dynamically analyze data dependencies between multiple nets. We believe such an algorithm can lay the foundation for uncovering data-level parallelism in routing: a necessary requirement for employing high throughput hardware. Detailed simulation results show an average of 4X speedup over NTHU-Route 2.0 with negligible loss in solution quality. To the best of our knowledge, this is the first work on utilizing GPUs for global routing.

I. Introduction

Global routing problem (GRP) is one of the most computationally intensive processes in VLSI design. Since the solution of the GRP is used to guide further optimizations before tape-out, it also becomes a critical step in the design cycle. Consequently, both the execution time and the solution quality of the GRP substantially affects the chip timing, power, manufacturability as well as the time-to-market.

Aggressive technology scaling introduces several additional constraints in the GRP, significantly increasing the complexity of this important VLSI design problem [7], [15]. Alpert et al. predicts that at 32nm there will be 4-6 metal widths and 20 thicknesses across 12 metal layers [1]. Furthermore, IBM envisions an explosion in design rules beyond 22nm that will make GRP a multi-objective problem [14]. Unfortunately, current CPU-based routers will prove to be inefficient for the increasingly complex GRP as these routers only solve simple optimization problems [9], [22].

Tackling this huge computationally complex problem would require a platform that offers high-throughput computing such as a *Graphics Processor Unit (GPU)*. Traditionally, a GPU's computing bandwidth is used to solve massively parallel problems. GPUs excel in applications that repeatedly apply a set of operations on a big data set, involving single instruction multiple data (SIMD) style parallel code. Several existing VLSI CAD problems have seen successful incarnation in GPUs, delivering more than 100× speedup [6], [10], [11]. However, the canonical GRP does not fit well into such an execution paradigm because routing algorithms repeatedly manipulate shared data structures such as routing resources. This sharing of resources disrupts the data-independence requirement of traditional GPU applications. Hence, existing task-based parallel routing algorithms must be completely revamped to make use of the GPU bandwidth.

In the light of these technology trends, we propose a hybrid GPU-CPU routing platform that enables a collaborative algorithmic framework to combine data-level parallelism from GPUs with thread-level parallelism from multicores. Our work specifically addresses the scalability challenges posed to current global routers. Till date, there has been very few works that parallelize the GRP by using multicore processors [17], [27]. However, none of these are designed to exploit high throughput computing platforms such as the GPU.

Exploiting the computation bandwidth of GPUs for the GRP is a non-trivial problem as the overhead of sharing resources hurts the overall performance. In this work, we use a fundamentally new mode of parallelism to uncover the performance potential of the GPU. We propose a novel *Net Level Concurrency (NLC)* model to *efficiently* consider the data dependencies among all simultaneously routed nets. This model enables parallelism to scale well with technology and computing complexity.

Following are the major contributions of our work to global routing research:

- **GPU-CPU Hybrid Routing:** We propose an execution model that allows cooperation of the GPU and the CPU to route multiple nets simultaneously through *Net Level Concurrency* (NLC). To the best of our knowledge, this is the first work on utilizing GPUs for global routing. The GPU global router uses a Breadth First Search (BFS) heuristic while the CPU router uses A* maze routing. Together, they provide two distinct classes in the routing spectrum. The high-latency low-bandwidth problems are tackled by the CPU, whereas the low-latency high-bandwidth problems are solved by the GPU. We believe this classification is the key to efficiently tackle the complexity increase of the GRP on massively parallel hardware.
- **Scheduler:** We develop a scheduler algorithm to explore NLC in the GRP. The scheduler produces concurrent routing tasks for the parallel global routers based on net dependencies. The produced concurrent tasks are distributed to the parallel environments provided by the GPU and multicore CPU platforms. The scheduler is designed to dynamically and iteratively analyze the net dependencies, hence limiting its computational overhead.
- **GPU breadth-first search:** We propose a breadth-first search based path finding algorithm on a GPU. This

978-1-4577-1400-9/11/$26.00 ©2011 IEEE

algorithm utilizes the massively parallel architecture for routing and back tracing. Our approach is able to find the shortest weighted path, and achieves high computational throughput by simultaneously routing multiple nets.

The remainder of this paper is organized as follows: In Section II, we briefly discuss related literature. Section III describes the overview of our GPU-CPU router. We discuss the challenges of a scalable parallel routing algorithm in Section IV. Section V presents our detailed *Scheduler* design, while our GPU-CPU router implementations are described in Section VI. We show our results in Section VII and we conclude in Section VIII.

II. Related Work

In 2007 and 2008, the International Symposium on Physical Design (ISPD) held two global router competitions. These contests promoted the development of many recent global routers. These routers typically employ a collection of well-studied routing techniques. Roughly, we can categorize them into two types: *sequential* and *concurrent*. The sequential techniques apply maze routing followed by a negotiation-based rip-up and reroute (RRR) scheme. RRR was originally introduced in Pathfinder for FPGAs to route macro cells [20]. Consequently, it has been used to recursively reduce overflows in the following routers: FGR [24], NTHU-Route 2.0 [2], NCTUgr [8], NTUgr [3] and Archer [23]. The concurrent algorithms apply Integer Linear Programming (ILP) to achieve an overflow-free solution [4], [26].

GRIP [26] currently holds the best solution quality in open literature of all ISPD 2007 and 2008 benchmarks. However, its pure ILP-based approach requires significantly longer runtime compared to the negotiation-based RRR scheme. Even the runtime reported in their parallel PGRIP [27] router was still relatively high.

When problems approach a larger scale, sequential global routers are more popular because the negotiation-based RRR scheme offers a much better trade-off between solution quality and runtime. This scheme can even be applied in addition to an ILP-based approach to reduce runtime [4]. However, the downside of the sequential approach is the heavy dependency of solution quality on the routing order of nets.

Due to this reason, parallelization of negotiation-based RRR scheme is difficult. Routing multiple nets simultaneously could jeopardize the routing order and create race conditions on shared routing resources among threads. Recently, Liu et al. adopted a collision-aware global routing algorithm with Bounded-Length maze routing on a quad-core system [17]. They implemented a rudimentary workload-based parallelization technique with simple collision-awareness algorithm. Their router has achieved good speedups on 4 threads.

Our work is the first to use a high-throughput hybrid environment (GPU-CPU) to tackle the GRP. As technology develops, the boundary between SIMD (single instruction multiple data) and SISD (single instruction single data) will soon shrink [25]. This work will lay the foundation for using hybrid high-throughput environment for global routing.

III. Overview of GPU-CPU Global Routing

In this Section, we give an overview of our GPU-CPU global router.

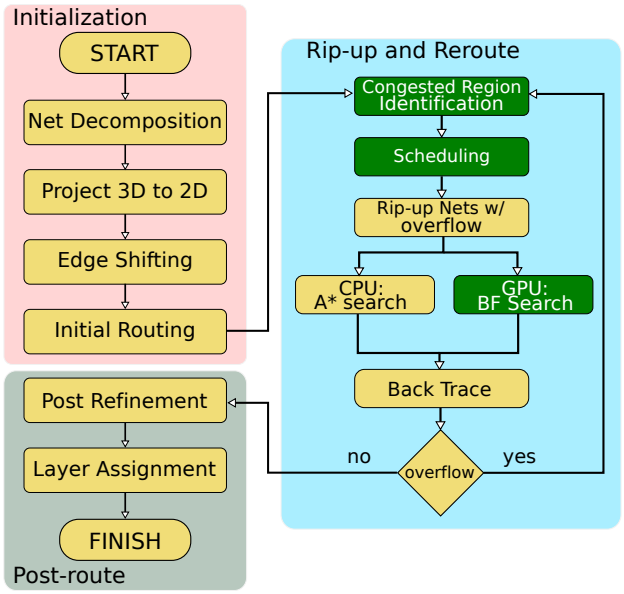


Fig. 1: Global router design flow. Top left section is initialization phase while bottom left is post-routing phase. Steps in these two sections are also present in other CPU-based routers. Right section is rip-up and reroute, we enhanced this section by using a scheduler. Our contributions are highlighted in dark background shading.

A. Objective

Like other global routers [2], [3], [4], [8], [18], [23], [24], [21], [28], [27], the GPU-CPU global router has three major objectives. First is the minimization of the sum of overflows among all edges. Second is to minimize the total wirelength of routing all the nets, and third is the minimization of the total runtime needed to obtain a *solution*.

B. Design Flow

The flow of our global router is shown in Figure 1. The initial global routing solution is generated as the following: we first project a multi-layer design on a 2D plane and use FLUTE 2.0 [5] to decompose all multi-pin nets into sets of two-pin subnets. Consequently, we apply edge shifting on all the nets to modify their initial topologies. We then perform initial routing and create the congestion map.

During the main phase, we apply negotiation-based scheme to iteratively improve the routing quality by ripping-up and rerouting the subnets that pass through overflowing edges. We route each subnet within a *bounding box*, whose size is relaxed if the subnet is unable to find a feasible path (Section VI-C). The order of the subnets to be ripped-up and rerouted is determined through *congested region identification*, an algorithm that collects subnets that are bounded within the congestion region (Section VI-D).

The main phase completes when no overflow is detected. Then we apply layer assignment to project the 2D routing plane back onto the original multi-layer design. The layer assignment technique is similar to that described in [24].

C. Global Routing Parallelization

The parallel global router strives for high throughput by maximizing the number of simultaneously routing nets. However, the negotiation-based RRR scheme is strongly dependent on the routing order. Routing nets simultaneously regardless

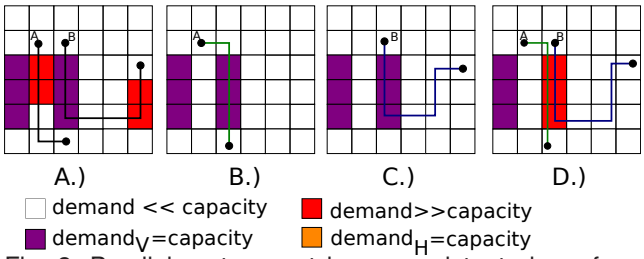


Fig. 2: Parallel router must have consistent view of resources. A.) Before rip-up and re-route B & C.) Viewpoint of each thread. They unknowingly allocate conflicted resources D.) An overflow is realized at the end of R&R when both threads back track.

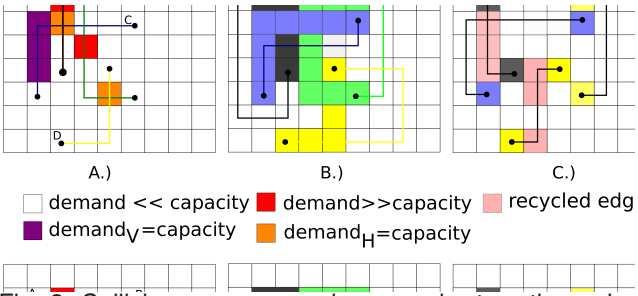


Fig. 3: Collision awareness alone can hurt routing solution. A.) 4-thread router processing a particular congested region, 1 net/thread B.) Routing solution generated via collision-aware algorithm. Some resources are wasted due to overhead of collision awareness because threads are discouraged to route on cells (black, green, yellow and blue cells) that were previously used by another thread. C.) With proper scheduling, only one thread is processing this particular region and some of the resources are recycled. Remaining threads are routing other congested areas in the chip (not shown).

of their order might cause degradation in solution quality and performance.

We tackle this problem by examining the dependencies among nets, and extracting concurrent nets from the ordered task queue. These nets can then be routed simultaneously without jeopardizing the solution quality. We now discuss challenges of extracting concurrent nets (Section IV), and tackling them through a novel *Scheduler* (Section V).

IV. Enabling Net Level Parallelism in Global Routing

In this Section, we will explain the requirements of enabling data-level parallelism in the GRP: a necessary step for exploiting high throughput platforms such as GPUs.

A. Challenge in Parallelization of Global Routing

There are two main approaches in parallelizing the global routing problem. First, the routing map can be partitioned into smaller regions and solved in a bottom-up approach using parallel threads [13], [16], [27]. Second, individual nets can be divided across many threads and routed simultaneously. We call this *Net Level Concurrency (NLC)*. NLC can substantially achieve better load-balancing and uncover more parallelism. However, it also comes at the cost of additional complexity.

Unlike partitioning, NLC allows sharing of routing resources between multiple threads. Consequently, to effectively exploit NLC, one must ensure that threads have current usage information of the shared resources. Without such information, concurrent threads may not be aware of impending resource collisions, leading to unintentional overflow and degradation in both performance and solution quality [17]. This phenomenon is demonstrated in Figure 2, where both the threads consume the lone routing resource resulting in an overflow.

B. Achieving NLC

Unfortunately, collision-awareness alone cannot guarantee reaping performance benefits by employing NLC on high throughput platforms. This is best explained through Figure 3. In this example, we assume that there is a task queue and that threads continually get a net to route from this queue. We also assume that nets in the queue are ordered based on the congestion of their path. Finally, there are two layers (horizontal and vertical) available for routing with demands indicated as $demand_H$ and $demand_V$. When multiple threads are processing a congested region, as in Figure 3B, concurrent routing will cramp limited resources to specific threads, sacrificing performance and solution quality. This problem is exacerbated with increasing number of threads and circuit complexity. However, we observe that we can recover substantial performance by allowing threads to concurrently route in different congested regions. In other words, a single thread can process the region shown in Figure 3C.

Therefore, the **key to achieving high throughput in NLC is to identify congested regions and co-schedule nets that have minimal resource collisions**. In the next Section, we will discuss our proposed *Scheduler* design.

V. Scheduler

In this Section, we describe our Scheduler that dynamically examines the dependencies among nets in an ordered task queue, and extracts concurrent nets to be dispatched to the GPU and CPU routers for parallel routing.

A. Nets Data Dependency

We create an explicit routing order for Rip-up and Re-Route (RRR) from all the nets found within the same congested region. The order is derived by a comparison of area and aspect ratio of the nets. Certain nets are given higher priority in this order (e.g., nets covering larger area with a smaller aspect ratio), based on the ease of identifying an overflow free path.

This routing order introduces data dependencies among nets. Each 2-pin net (subnet) has a bounding region that constrains its demand for routing resources. We define *independent subnets* as subnets with no overlapping bounding region, as these can be routed simultaneously. Subnets with intersecting bounded regions cannot be routed concurrently due to shared resources. The data dependency among them dictates that the original routing order needs to be preserved in order to achieve the same routing quality.

Honoring the routing sequence due to net dependencies is crucial for our parallel model. The existing task-based parallel global router does not examine the net dependency when exploiting concurrency [17]. As a result, more than 41% of the subnets are affected by collisions in shared routing resources. This model does not suit well in a GPU-based concurrency framework. Due to the lack of synchronization mechanism for thread blocks in the GPU hardware, we need to avoid resource collision on the GPU's device memory.

In this paper, we propose a *Scheduler* to generate independent routing tasks for the parallel global routers. The data dependency is iteratively analyzed, thereby limiting its analysis

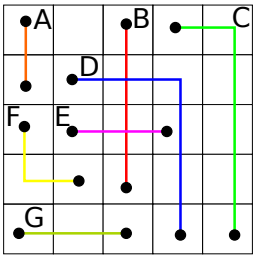
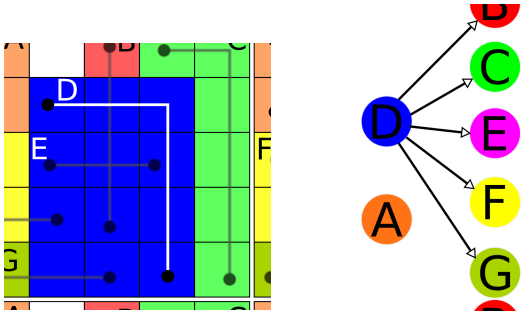


Fig. 4: Routing problem with nets overlapping each other



(a) Colored Tiles (b) Dependency Graph

Fig. 5: Results after the 1st iteration (a) coloring of tiles: bigger nets dominate ownership over smaller ones. Only A and D can be routed together because other nets are dependent on D. (b) Net dependencies are derived from the colormap.

overhead while providing precise dependency information. First, the data dependency among nets is constructed in a dependency graph. Then we exploit parallelism by routing the independent nets. The parallelism created by our model can exploit massively parallel hardware without causing resource collision or jeopardizing the routing order. As a result, our GPU-CPU parallel global router achieves *deterministic* routing solutions.

B. Net Dependency Construction

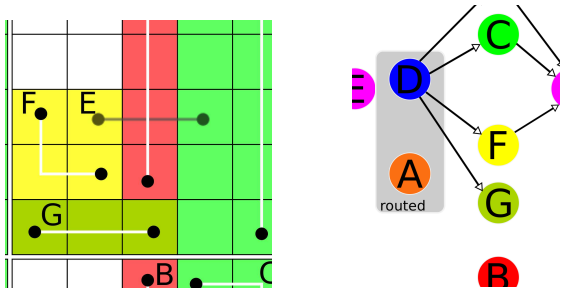
In this Subsection, we present the scheduler algorithm to construct the subnet dependencies. As an example, a selection of 2-pin nets (subnets) are illustrated in Figure 4. In this example we assume that each subnet's bounding region is the same size as their covering area, and that the routing order derived is:

$$D > C > F > B > E > G > A$$

We now explain how to construct the dependency graph, and exploit the available concurrency without causing conflict in routing resource or order. This approach is mainly divided into the following 3 steps.

Tile Coloring: In this step, each tile identifies its occupancy by iterating through the ordered subnets. The first subnet region that covers the tile is considered as its occupant. Using the example from Figure 4, the results of the colored tiles are shown in Figure 5(a). We can observe that most of the map is colored by subnet D because it has the highest priority in routing order. Given this colormap, each subnet can visualize the other subnets that it is overlapping with, hence determining its dominant subnets.

Finding Available Concurrency: With the help of the colormap, we can easily find subnets that can be routed by



(a) Colored Tiles (b) Dependency Graph

Fig. 6: Results after the 2nd iteration (a) After D and A are routed, nets C, B, F and G can be routed together because they have no dependencies (b) More detailed dependencies are revealed in the graph.

checking if it occupies all of its routing regions. If not, then there is a dependency on other subnets and it must wait until the dependency is resolved. In Figure 5(a), subnets A and D occupy all of their routing regions. Hence, they can be scheduled together.

Now we introduce the concept of **dependency level**. This metric is used to determine the urgency of routing certain subnets. We score the dependency level as the number of routing regions that one subnet invades. For instance, in Figure 5(a) subnet D scores 5 because it invades the area of 5 subnets: B, C, E, F, and G.

From Figure 5(b) we can make an interesting observation on the dependencies among all subnets. Subnets B, C, E, F, and G are dependent on subnet D but we cannot identify the dependencies between them. The algorithm intentionally leaves these detailed dependencies for future computation, so as to reduce complexity while extracting the available concurrency in a timely manner.

Tile Recoloring: In this step, the algorithm uncovers detailed dependencies by reconstructing the colormap. After the scheduled subnets are routed, the colormap must be reconstructed to resolve previous dependencies. Figure 6(a) shows the new colormap when subnets A and D are already routed. Recoloring only needs to consider subnets that were dominated by A and D. In this case, they are $\{C, F, B, E, G\}$ for D's region, and $\emptyset$ for A.

The dependency graph is updated using the new map. Figure 6(b) shows the new graph that reveals a new dependency between subnets B, E, C, and F. Similar to the previous iteration, the dependency graph indicates subnets B, C, F, and G can be scheduled once subnet D has finished, while E can only be scheduled after B, C, and F are completed.

C. Implementation and Optimization

The above three steps are recursively applied until all dependencies are resolved. *The scheduler thread and the global router threads execute in parallel with a producer-consumer relationship.* The scheduler keeps constructing the dependency graph with the given task queue, and producing concurrent nets to the router threads. The routers consume these nets simultaneously with different priorities indicated by the dependency levels, and return the completed nets to release more concurrent nets.

The efficiency of the scheduler algorithm affects the available computational throughput for routing. The complexity of this algorithm increases as the region size and net count

increases. In practice, we reduce the problem size using the identified congestion region (Section VI-C). Instead of exploring concurrency on the entire routing map, the search area is restricted to only the congestion regions, hence easing the computation load of tile coloring.

In addition, a *task window* is used to restrict the number of nets being examined for concurrency in each iteration. In some cases, congestion regions contain a large number of nets. The task window can effectively limit the search space and speed-up the dependency tree construction.

VI. Implementation

In this Section, we will discuss details of our GPU-CPU router implementation. We focus on several key issues such as data representation in the GPU (VI-A), efficient search techniques in the GPU (VI-B), directional congested region identification algorithm (VI-C), our bounding box expansion method (VI-D), and distribution of nets among CPU and GPU threads (VI-E).

A. GPU Data Structures

Routing algorithm requires efficient data structures for graph representation and traversal. Unlike allocating user-defined data structures in the system memory, the GPU offers much less flexibility in dynamic device memory allocation. Creating an efficient data structure on the GPU that is suitable for routing is a challenging task. We will explain how we took advantage of inherent technological features of a CUDA-enabled GPU to optimize our data structures for routing.

We store the vertices topology and edge cost data in the GPU texture memory. This memory is optimized for read-only data in the global memory. Fetching from cached texture memory provides high bandwidth on memory space with good spatial locality (i.e. if a cell is visited, then its neighbors are also traversed). Hence, we bind the vertices and edge costs array with `3DTexture` and `1DTexture` memories, respectively. Each cell in the vertices in `3DTexture` points to six different adjacent cells: `-X, -Y, -Z, +X, +Y, +Z`. Significant performance benefit is seen using 3D texture fetch.

All costs of traversing tiles are arranged in 2D arrays. We store these data on the local shared memory in each thread block rather than on the global device memory. This arrangement has a much higher access efficiency, but comes at the cost of generality. Due to the size limitation of the shared memory on a GPU (48 KB on Fermi), the number of tiles that can be traversed by each thread block is constrained to about 2500. Fortunately, this size is reasonable for the GRP in most cases. According to our observation, more than 99% of all 2-pin nets can be fitted within this area.

B. Pathfinding Implementation in GPU

The GPU architecture has shown great potential in accelerating performance of data-intensive computations. For an application to harness the power of the GPU, it must obey the SIMD model where a specific set of operations are applied repeatedly to a large chunk of data. Unfortunately, most of the path finding algorithms used in CPU routers are optimized for sequential execution and are not amenable to parallelization. In addition, CPU-based maze routing algorithms, i.e. A* search, Dijkstra's algorithm, use dynamic data structures such as heap

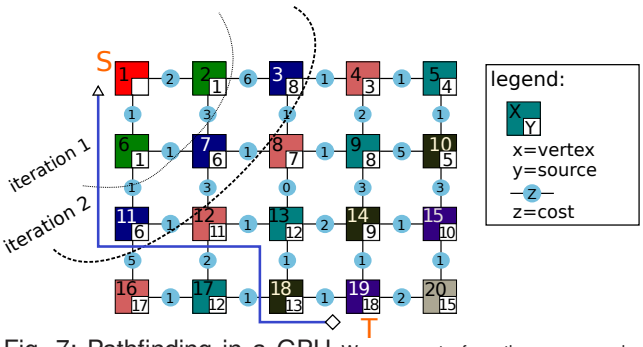


Fig. 7: Pathfinding in a GPU We propagate from the source node. The breadth-first search fills up the entire search region, and continues until all frontiers are exhausted. Then we back trace from the target node to find the shorted weighted path.

to keep track of all traversing paths. These data structures are too complex for the GPU architecture to maintain. For these reasons, most CPU-based maze routing algorithms are not suitable for the GPU architecture.

Algorithm 1 GPU BFS Algorithm Kernel

```
1: tid ←getThreadID
2: if Frontier[tid] then
3:    Frontier[tid] ←false
4:    for all neighbors nid of tid do
5:       Edge ← Vertex(tid, nid)
6:       if Edge exists then
7:          AddedCost ← TileCost[tid] + EdgeCost[Edge]
8:          if AddedCost < TempTileCost[nid] then
9:             TempTileCost[nid] ← AddedCost
10:         end if
11:      end if
12:   end for
13: end if
14: SYNCHRONIZE_THREADS()
15: if TileCost[tid] > tempTileCost[tid] then
16:    TileCost[tid] ← tempTileCost[tid]
17:    Frontier[tid] ← true
18:    DONE ← false
19: end if
20: tempTileCost[tid] ← TileCost[tid]
```

Our GPU router uses parallel breadth-first search (BFS) algorithm to find weighted shortest paths. Like other BFS algorithms for GPUs [12], [19], our GPU-BFS exploits parallelism by simultaneously exploring vertices at the same depth, which are defined as *frontiers*. In each iteration, all frontiers add their current costs to the costs of edges that connect to the neighboring tiles. If the sum results in a lower cost on a neighboring tile, then this tile is activated as a new frontier. The previous frontier is then deactivated. The search stops when the frontiers are all exhausted, hence guarantees all possible paths within the search region are traversed.

Our algorithm is fundamentally different from the previously proposed BFS algorithms for GPUs [12], [19], since a) our algorithm tackles the weighted shortest path problem; b) we route an individual 2-pin net within each thread block. Therefore, we attain performance boost by routing large amounts of nets concurrently.

Term	Description
$Frontier$	Boolean list that marks the frontier tiles in the current iteration. Contains source vertex initially.
$EdgeCost$	Array that stores cost of all edges.
$Vertex$	Function that returns the $Edge$ between two Vertices.
$AddedCost$	Intermediate variable to store the new tile cost.
$TempTileCost$	Array that stores cost of traversing tiles. Initialized as Inf.
$TileCost$	Array that stores minimum cost of traversed tiles. Initialized as Inf.
$DONE$	Boolean indicating all frontiers are explored.

TABLE I: BFS Notations

Details of our parallel BFS are presented in Algorithm 1, with the explanation of terms in Table I. Figure 7 shows an illustrative example for this work.

In the first part of the kernel, each thread checks its entry in the frontier array $Frontier$. It updates the cost of its neighbors by adding its own cost and the edge cost. Each thread also removes its vertex from $Frontier$ and adds the new cost to the traversing tiles in $tempTileCost$. Here all threads must synchronize to remove read/write inconsistencies in the shared memory. The second part of the kernel copies the contents from $tempTileCost$ to $TileCost$ if a lower cost is found, and marks these tiles as new paths in the $Frontier$. The process is repeated until all frontiers are explored. The path is then generated using a back trace algorithm in the GPU. The back tracing (not shown) essentially reverses the searching order of the above algorithm.

C. Congested Region Identification (CRI)

The first step of identifying a congested region is to recognize the most congested tile during routing. Subsequently, a region is marked as congested by expanding around the congested tile.

Term	Description
$r_{i,j;k,l}$	rectangle with bottom left coordinates (i, j) and top right (k, l)
$Ave_i(r)$	average congestion value inside the expanded rectangle r in the direction of i side(s)
$\mathcal{L}_{\mathcal{B}}(level)$	returns the lower bound value for a particular congestion level
$expand4sides(r)$	expands region r in all directions
$expand3sides(r)$	expands 3 sides of region r towards the maximum congestion
$expand2sides(r)$	expands 2 sides of region r towards the maximum congestion
$expand1sides(r)$	expands 1 side of region r towards the maximum congestion

TABLE II: Algorithm Notations

In order to accurately identify the congested region, we use a *directional expansion algorithm* to adaptively expand to the region in the directions that result in the highest congestion. Figure 8 best explains this situation. We begin by taking the most congested tiles (red cells in the figure) and adaptively expand until the average congestion for the region is below a certain threshold. We divide the congestion value into several congestion levels, much like NTHU-Route [2]. The number of congestion levels we seek to model will dictate the size of each region. In our example, we have 4 congestion levels. After these regions are found, we route the nets inside them in parallel based on Section IV. Pertinent details are presented in Algorithm 2, Table II lists the notations and their descriptions.

Algorithm 2 Directional Expansion Algorithm

$r = max(b_{ij})$
$\forall\ 0 \leq i \leq X; 0 \leq j \leq Y$

```
for level = 1 to 4 do
    while Ave_4(r) > L_B(level) do
        expand4sides(r)
    end while
    while Ave_3(r) > L_B(level) do
        expand3sides(r)
    end while
    while Ave_2(r) > L_B(level) do
        expand2sides(r)
    end while
    while Ave_1(r) > L_B(level) do
        expand1sides(r)
    end while
end for
```

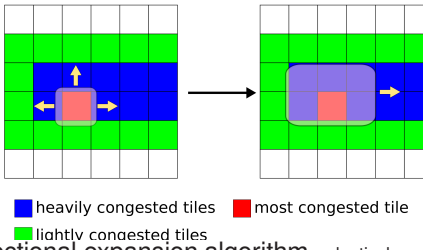


Fig. 8: Directional expansion algorithm adaptively expands in the directions with the highest congestion.

D. Bounding Box Expansion

Bounding box is widely applied in global routers. This technique constrains the searching region of 2-pin nets within a rectangle, therefore reduces the space complexity of maze routing, and decreases the solution wirelength.

Typically, the size of the bounding box is expanded as the RRR iteration proceeds. As the remaining overflow decreases, we can relax the constraint of the bounding box and allow the maze router to obtain overflow free route at the cost of longer wirelength. But as the constraint of bounding box relaxes to more than 10 times as large as the original, we stop the expansion again to avoid excessively long routes.

We choose an adaptive method, rather than a fixed parameter function, to expand the bounding box. The search area constraint is relaxed accordingly to the percentage of remaining

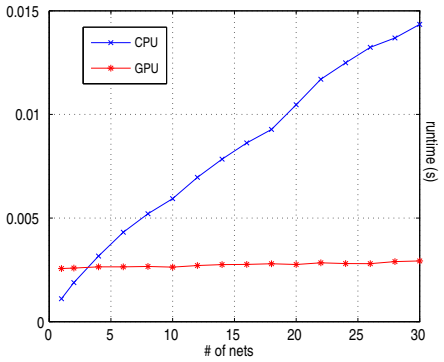


Fig. 9: Runtime comparison between CPU A*Search and GPU BFS.

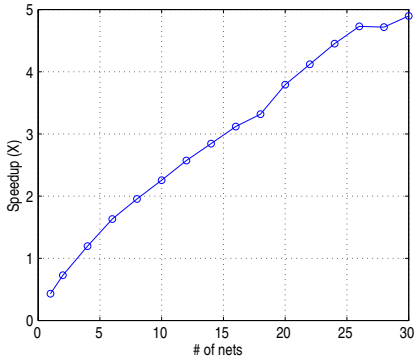


Fig. 10: Speedup of GPU BFS over CPU A*Search.

overflow. We keep the size of the bounding box unchanged until 99% of all overflow is resolved. This phase passes very fast since the searching areas are small. Then we linearly increase the size of bounding box as RRR iteration proceeds, until its size reaches the upper limit.

E. Workload Distribution between GPU and CPU

The scheduler dispatches workloads among the CPUs and GPU for optimum computational throughput. Typically, the CPU routers achieve a single solution with lower latency than the GPU router, but the latter can achieve much higher throughput by routing multiple nets in a single kernel call. Urgent nets, which release several subsequent nets to be routed concurrently, are scheduled on the CPUs. In addition, nets with large bounding boxes are also routed in the CPU due to the shared memory limits on the GPU.

VII. Results

The global router is implemented in C/C++ on an Intel Quad-core 2.4GHz machine with 4GB of RAM. The GPU in this study is an Nvidia Geforce GTX 470, featuring the Fermi architecture. The GPU code is compiled with the CUDA Toolkit 4.0RC2. We use ISPD 2007 and ISPD 2008 benchmark suites in the experiments shown below.

A. GPU and CPU router

We first compare the routing throughput between our GPU and CPU routers. In this experiment, we schedule the same nets to the GPU router and a single CPU router, and record the average wall clock time for both routers. To make it a fair comparison, the process includes routing, back tracing, and data transfers between the GPU and the system memory.

The results are shown in Figure 9 and Figure 10. The runtime comparison in Figure 9 shows a linear increase of CPU router runtime with the growing number of routing nets. Interestingly, the GPU runtime slope is much flatter than the CPU. Consequently, CPU router has a much shorter latency when routing individual nets, while the GPU can deliver a much higher bandwidth when scheduled with multiple nets.

The relative speedup between the GPU and CPU routers is illustrated in Figure 10. We can observe about 5X speedup when both the routers are scheduled with 30 nets. One should notice that the speedup keeps growing with more scheduled nets. We have observed a speedup of 73X if both routers are scheduled with 1000 nets (not shown).

B. Comparison with NTHU-Route 2.0

We now compare our parallel router with the ISPD 2008 routing contest winner NTHU-Route 2.0. The experiments are run on the same hardware platform. The results of the overflow free benchmarks are listed in Table III.

Our parallel router generates high quality routing solutions with wirelength within an average of 1.1% increase to that reported by NTHU-Route 2.0. Noticeably, the additional GPU router introduces negligible overhead in the resultant wirelength. In the runtime comparison, the parallel router utilizing 4 CPU threads achieves an average speedup of 3.34X compared to NTHU-Route 2.0, while nearly 4X (3.96X) average speedup is achieved with the additional GPU router. These results prove the effectiveness of our concurrency model in solving GRP with NLP on a high throughput hardware, while the concurrency has little affect in the solution quality.

Although the GPU shows great potential in offering high throughput in routing nets (Figure 10), our observed speedup is somewhat less. The GPU heavily depends on the *Scheduler* to uncover independent nets that can be routed. In our experiments, we have rarely observed more than 30 concurrently routable nets. In some situations, the insufficient amount of independent nets can even cause the GPU router to wait in idle, or to be configured with suboptimal number of nets. Nevertheless, our proposed algorithm on dynamically detecting data level parallelism in routing lays the foundation for future research to exploit GPUs for high speed routing.

VIII. Conclusion

As technology continues to scale, computational complexity of many EDA algorithms is growing rapidly. Exploiting the computational bandwidth of high throughput platforms like the GPU is a prominent direction for future EDA. In this paper, we present a hybrid GPU-CPU high throughput computing environment as a scalable alternative to the traditional CPU based router. We show that the traditional GRP needs to be revamped for exploiting the new computing environment. The key to our method is using *Net Level Concurrency* guided by a Scheduler. The Scheduler analyzes data dependencies between nets and dynamically generates concurrent routing tasks for the computing environment. Detailed simulation results show an average of **4**X speedup over NTHU-Route 2.0 with negligible loss in solution quality. Our framework is a concrete step

	Parallel Router(4-core)		Parallel Router(4-core+GPU)		NTHU 2.0		Speedup	
	WL[a]	runtime[b]	WL[a]	runtime[b]	WL[a]	runtime[b]	4-core	4-core+GPU
adaptec1	5.43E6	2.90	5.44E6	2.07	5.34E6	9.95	3.43	4.81
adaptec2	5.29E6	0.70	5.30E6	0.63	5.23E6	2.1	3.00	3.33
adaptec3	1.31E7	3.53	1.31E7	3.05	1.31E7	10.86	3.08	3.56
adaptec4	1.24E7	0.95	1.24E7	0.65	1.22E7	2.5	2.63	3.85
adaptec5	1.55E7	9.98	1.55E7	8.12	1.55E7	21.9	2.19	2.70
newblue1	4.70E6	2.87	4.70E6	1.88	4.65E6	6.2	2.16	3.30
newblue2	7.79E6	0.28	7.81E6	0.27	7.57E6	1.1	3.93	4.07
newblue5	2.38E7	5.38	2.38E7	4.42	2.32E7	19.1	3.55	4.32
newblue6	1.80E7	4.12	1.80E7	3.78	1.77E7	17.5	4.25	4.63
bigblue1	5.63E6	3.10	5.63E6	2.86	5.59E6	13.1	4.22	4.58
bigblue2	9.10E6	2.31	9.05E6	2.09	9.06E6	8.4	3.64	4.02
bigblue3	1.30E7	1.09	1.30E7	1.01	1.31E7	4.4	4.04	4.37

a. wirelength in terms of edges consumed
b. expressed in minutes

TABLE III: Wirelength and runtime comparison with NTHU-Route 2.0.

towards developing next generation global routers geared for high throughput compute architectures.

Acknowledgements

This work was supported in part by National Science Foundation grant CNS-1117425.

REFERENCES

[1] ALPERT, C. J., LI, Z., MOFFITT, M. D., NAM, G.-J., ROY, J. A., AND TELLEZ, G. What makes a design difficult to route. In *Proc. of ISPD* (2010), pp. 7–12.

[2] CHANG, Y.-J., LEE, Y.-T., AND WANG, T.-C. NTHU-Route 2.0: A Fast and Stable Global Router. In *Proc. of ICCAD* (2008), pp. 338–343.

[3] CHEN, H.-Y., HSU, C.-H., AND CHANG, Y.-W. High-performance global routing with fast overflow reduction. In *Proc. of ASP-DAC* (2009), pp. 570–575.

[4] CHO, M., LU, K., YUAN, K., AND PAN, D. Z. BoxRouter 2.0: Architecture and Implementation of a Hybrid and Robust Global Router. In *Proc. of ICCAD* (2007), pp. 503–508.

[5] CHU, C., AND WONG, Y.-C. FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design. *TCAD 27*, 1 (Jan. 2008), 70–83.

[6] CROIX, J., AND KHATRI, S. Introduction to GPU programming for EDA. In *Proc. of ICCAD* (nov. 2009), pp. 276 –280.

[7] CROSS, D., NEQUIST, E., AND SCHEFFER, L. A DFM aware, space based router. In *Proc. of ISPD* (2007), pp. 171–172.

[8] DAI, K.-R., LIU, W.-H., AND LI, Y.-L. Efficient simulated evolution based rerouting and congestion-relaxed layer assignment on 3-D global routing. In *Proc. of ASP-DAC* (2009), pp. 582–587.

[9] GROENEVELD, P. Going with the flow: bridging the gap between theory and practice in physical design. In *Proc. of ISPD* (2010), pp. 3–3.

[10] GULATI, K., AND KHATRI, S. Towards acceleration of fault simulation using Graphics Processing Units. In *Proc. of DAC* (june 2008), pp. 822 –827.

[11] GULATI, K., AND KHATRI, S. P. Accelerating statistical static timing analysis using graphics processing units. In *Proc. of ASP-DAC* (2009), IEEE Press, pp. 260–265.

[12] HARISH, P., AND NARAYANAN, P. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *High Performance Computing* (2007), pp. 197–208.

[13] HU, J., AND SAPATNEKAR, S. S. A Survey on Multi-Net Global Routing for Integrated Circuits. *Integration, the VLSI Journal 31* (2001), 1–49.

[14] JOHNSON, R. C. EE Times: IBM warns of 'design rule explosion' beyond 22-nm, Mar. 2010.

[15] KAUL, N. Design planning trends and challenges. In *Proc. of ISPD* (2010), pp. 5–5.

[16] KHANNA, S., GAO, S., AND THULASIRAMAN, K. Parallel hierarchical global routing for general cell layout. In *Proceedings of the Fifth Great Lakes Symposium on VLSI (GLSVLSI'95)* (Washington, DC, USA, 1995), GLSVLSI '95, IEEE Computer Society, pp. 212–.

[17] LIU, W.-H., KAO, W.-C., LI, Y.-L., AND CHAO, K.-Y. Multi-threaded collision-aware global routing with bounded-length maze routing. In *Proc. of DAC* (2010), pp. 200–205.

[18] LIU, W.-H., KAO, W.-C., LI, Y.-L., AND CHAO, K.-Y. Multi-Threaded Collision-Aware Global Routing with Bounded-Length Maze Routing. In *Proc. of DAC* (Anaheim, California, June 2010), pp. 200–205.

[19] LUO, L., WONG, M., AND MEI HWU, W. An Effective GPU Implementation of Breadth-First Search. In *Proc. of DAC* (Anaheim, California, June 2010), pp. 52–55.

[20] MCMURCHIE, L., AND EBELING, C. PathFinder: a negotiation-based performance-driven router for FPGAs. In *Proc. ACM third international symposium on Field-programmable gate arrays* (Feb. 1995), pp. 111–117.

[21] MOFFITT, M. D. MaizeRouter: Engineering an Effective Global Router. *IEEE Trans. of CAD 27*, 11 (Nov. 2008), 2017–2026.

[22] MOFFITT, M. D., ROY, J. A., AND MARKOV, I. L. The coming of age of (academic) global routing. In *Proc. of ISPD* (2008), pp. 148–155.

[23] OZDAL, M. M., AND WONG, M. D. F. Archer: a history-driven global routing algorithm. In *Proc. of ICCAD* (San Jose, California, Nov. 2007), pp. 488–495.

[24] ROY, J. A., AND MARKOV, I. L. High-performance Routing at the Nanometer Scale. In *Proc. of ICCAD* (San Jose, California, Nov. 2007), pp. 496–502.

[25] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph. 27* (August 2008), 18:1–18:15.

[26] WU, T.-H., DAVOODI, A., AND LINDEROTH, J. T. GRIP: Scalable 3D Global Routing Using Integer Programming. In *Proc. of DAC* (2009), pp. 320–325.

[27] WU, T.-H., DAVOODI, A., AND LINDEROTH, J. T. A parallel integer programming approach to global routing. In *Proc. of DAC* (2010), pp. 194–199.

[28] XU, Y., ZHANG, Y., AND CHU, C. FastRoute 4.0: Global Router with Efficient Via Minimization. In *Proc. Asia and South Pacific Design Automation Conference* (2009), pp. 576–581.